SQL and R Integration for Inventory Analysis: A Case Study of R-Rated Movies in DVD Rental Stores

Anju Sambasivan

Introduction

In this project, we are analyzing data from a DVD rental company to learn how to use SQL for extracting and managing information. The main goal is to check how many R-rated movies are available in each store and compare the inventory between the stores. We will use SQL to get this data and create a bar chart in R to visualize it.

To do this, we will write an SQL query that uses important techniques like joining multiple tables, grouping data, and counting the number of movies. These features help us get the exact data we need from the database. We will also use common table expressions (CTEs) to break the query into smaller parts, making it easier to understand.

Once we have the data, we will create a bar chart in R to compare the inventory of R-rated movies in the two stores. This will give us a clear picture of how the stock is distributed. Additionally, we will look at the SQL query's execution plan to see how it works and find ways to make it run more efficiently if needed.

This project is a simple but effective way to learn how SQL and R work together for analyzing and visualizing data. By the end, we will understand how to use SQL for solving real-world business problems in data science.

```r
# Install and load necessary packages
library(RPostgres)
library(ggplot2)
```

- **`library(RPostgres)`**: This loads the RPostgres package, which allows us to connect to and interact with a PostgreSQL database. It's like a tool that helps R talk to the database to fetch and send data.

- `library(ggplot2)`: This loads the ggplot2 package, a powerful tool in R for creating visualizations such as bar charts, line graphs, and more. It's used to make the data easier to understand by presenting it in graphical form.

```r
# Establish a connection to the PostgreSQL database
con <- dbConnect(
  Postgres(),
  dbname = "Jeff",
  host = Sys.getenv("PG_HOST"),
  port = Sys.getenv("PG_PORT"),
  user = Sys.getenv("PG_USR"),
  password = Sys.getenv("PG_PASS")
)
```

- `dbConnect`: Creates the connection to the database so R can communicate with it.
- `Postgres()`: Tells R we are using a PostgreSQL database.
- **Details of the Database**:

1. `dbname`: The name of the database is "Jeff."
2. `host`, `port`, `user`, `password`: These are the server address, port number, username, and password required to log in.

- `Sys.getenv()`: Retrieves the login details (like the database host and password) from a secure file on the computer instead of writing them directly in the script. This keeps sensitive information private. The result is a connection object called `con`, which allows us to run queries and retrieve data from the database.
- The .Renviron file is used to securely store database connection details like host, port, username, and password. Instead of typing these directly in the R script, they are fetched dynamically using Sys.getenv(), ensuring sensitive information is not exposed in the code.
- Use this command in R to open or create the .Renviron file: `usethis::edit_r_environ()`. If the .Renviron file doesn't exist, this command will create a new one in home directory. In the text editor that opens, add key-value pairs for your environment variables

Why Use This Code?

- **Security**: Keeps passwords and other sensitive details safe.
- **Convenience**: Makes it easier to reuse the connection without re-entering details.
- **Functionality**: Enables us to fetch and analyze data directly from the database for the project.

```r
# Validate the connection
dbIsValid(con)
```

```
[1] TRUE
```

The function **`dbIsValid(con)`** checks if the database connection is still active and working properly.

- **`con`**: This is the connection object created using `dbConnect()`. It represents the link between R and the database.
- **`dbIsValid()`**: This function returns: `TRUE` if the connection is valid and still active. `FALSE` if the connection is no longer valid, meaning you can't use it to interact with the database anymore.

```r
# List tables and fields
dbListTables(con)
```

```
 [1] "actor"                      "address"
 [3] "category"                   "city"
 [5] "country"                    "customer"
 [7] "film"                       "film_actor"
 [9] "film_category"              "inventory"
[11] "language"                   "payment"
[13] "rental"                     "staff"
[15] "store"                      "actor_info"
[17] "customer_list"              "film_list"
[19] "nicer_but_slower_film_list" "sales_by_film_category"
[21] "sales_by_store"             "staff_list"
```

```r
dbListFields(con, "rental")
```

```
[1] "rental_id"    "rental_date"  "inventory_id" "customer_id"  "return_date"
[6] "staff_id"     "last_update"
```

```r
dbGetQuery(con, "SELECT * FROM rental LIMIT 10")
```

```
   rental_id         rental_date inventory_id customer_id         return_date
1          2 2005-05-24 22:54:33         1525         459 2005-05-28 19:40:33
2          3 2005-05-24 23:03:39         1711         408 2005-06-01 22:12:39
3          4 2005-05-24 23:04:41         2452         333 2005-06-03 01:43:41
4          5 2005-05-24 23:05:21         2079         222 2005-06-02 04:33:21
5          6 2005-05-24 23:08:07         2792         549 2005-05-27 01:32:07
6          7 2005-05-24 23:11:53         3995         269 2005-05-29 20:34:53
7          8 2005-05-24 23:31:46         2346         239 2005-05-27 23:33:46
8          9 2005-05-25 00:00:40         2580         126 2005-05-28 00:22:40
9         10 2005-05-25 00:02:21         1824         399 2005-05-31 22:44:21
10        11 2005-05-25 00:09:02         4443         142 2005-06-02 20:56:02
   staff_id         last_update
1         1 2006-02-16 02:30:53
2         1 2006-02-16 02:30:53
3         2 2006-02-16 02:30:53
4         1 2006-02-16 02:30:53
5         1 2006-02-16 02:30:53
6         2 2006-02-16 02:30:53
7         2 2006-02-16 02:30:53
8         1 2006-02-16 02:30:53
9         2 2006-02-16 02:30:53
10        2 2006-02-16 02:30:53
```

```r
query <- "SELECT COUNT(*) FROM rental"
dbGetQuery(con, query)
```

```
  count
1 16044
```

These commands allow us to explore the database structure. Understand what kind of data is stored in the `"rental"` table. Fetch specific data for further analysis, like a preview of rows or the total number of records.

- **`dbListTables(con)`**: Lists all the tables available in the database. Helps us see the different data structures stored in the database.
- **`dbListFields(con, "rental")`**: Lists all the column names (fields) in the table called `"rental"`. This helps us understand what kind of information the `"rental"` table contains.
- **`dbGetQuery(con, "SELECT * FROM rental LIMIT 10")`**: Runs a query to fetch the first 10 rows of the `"rental"` table. Provides a quick preview of the data to understand its content.

- **query <- "SELECT COUNT(*) FROM rental"**: Saves a query to count the total number of rows in the `"rental"` table. This helps us understand the size of the table.
- **dbGetQuery(con, query)**: Executes the saved query to count the rows in the `"rental"` table. Returns the total number of records in the table.

Calculating and Displaying Daily Income from Movie Rentals

```
dbGetQuery(con, "SELECT * FROM payment LIMIT 5")
```

```
  payment_id customer_id staff_id rental_id amount        payment_date
1      17503         341        2      1520   7.99 2007-02-15 22:25:46
2      17504         341        1      1778   1.99 2007-02-16 17:23:14
3      17505         341        1      1849   7.99 2007-02-16 22:41:45
4      17506         341        2      2829   2.99 2007-02-19 19:39:56
5      17507         341        2      3130   7.99 2007-02-20 17:31:48
```

```
dbListFields(con, "payment")
```

```
[1] "payment_id"   "customer_id"  "staff_id"     "rental_id"    "amount"
[6] "payment_date"
```

```
# Daily income analysis
query1 <- "
  SELECT
    DATE(payment_date) AS rental_date,
    SUM(amount) AS total_income
  FROM payment
  GROUP BY rental_date
  ORDER BY rental_date
"
```

- **SELECT DATE(payment_date) AS rental_date**: Extracts only the date (not the time) from the `payment_date` column. Renames it as `rental_date` for easier interpretation.
- **SUM(amount) AS total_income**: Adds up the payment amounts (`amount`) for each date. Renames the total as `total_income` to represent the daily revenue.
- **FROM payment**: Specifies that the data is being retrieved from the `payment` table.
- **GROUP BY rental_date**: Groups the data by each unique rental date. This ensures that all payments made on the same date are combined.

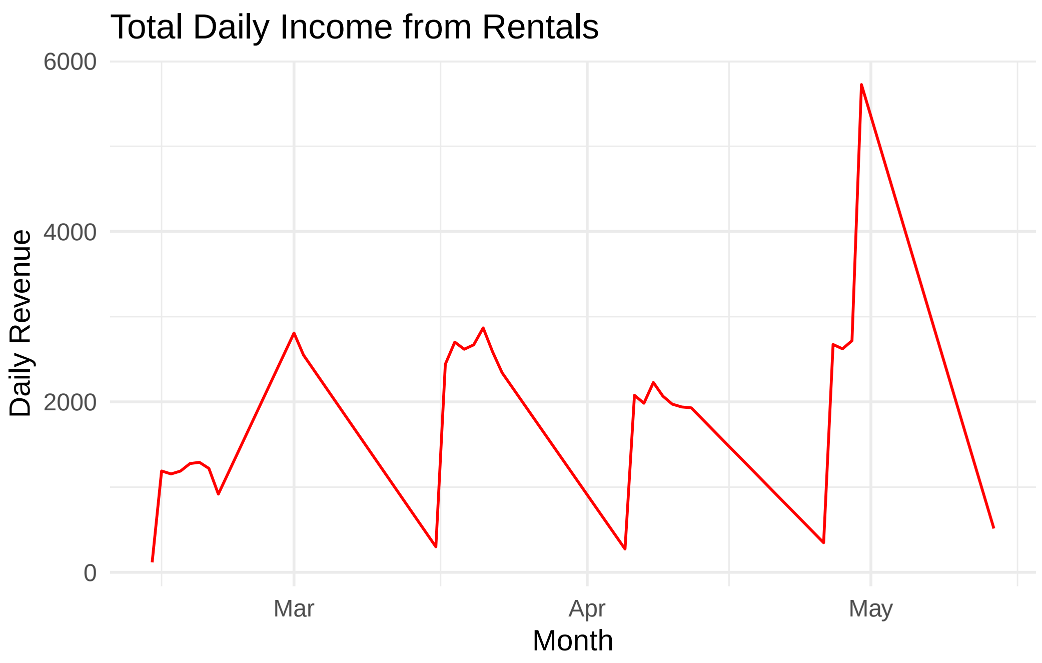
- **ORDER BY rental_date**: Sorts the results by the rental date, showing the income in chronological order.

```r
daily_income <- dbGetQuery(con, query1)
```

```r
head(daily_income)
```

```
  rental_date total_income
1  2007-02-14       116.73
2  2007-02-15      1188.92
3  2007-02-16      1154.18
4  2007-02-17      1188.17
5  2007-02-18      1275.98
6  2007-02-19      1290.90
```

```r
ggplot(daily_income, aes(x = rental_date, y = total_income)) +
  geom_line(color = "red") +
  labs(title = "Total Daily Income from Rentals", x = "Month", y = "Daily Revenue") +
  theme_minimal()
```



The graph shows how much money the company earns from movie rentals each day over three months (March, April, and May). The income goes up and down regularly, with a big spike in May followed by a sharp drop. These changes could be because more people rent movies on weekends, holidays, or during special events, while fewer people rent on regular weekdays. This information can help the company prepare for busy times, like keeping more DVDs ready when demand is high, and find ways to increase rentals on slower days.

```r
# address table
dbGetQuery(con, "SELECT * FROM address LIMIT 5")
```

```
  address_id              address address2 district city_id postal_code
1          1    47 MySakila Drive     <NA>  Alberta     300
2          2   28 MySQL Boulevard     <NA>      QLD     576
3          3    23 Workhaven Lane     <NA>  Alberta     300
4          4 1411 Lillydale Drive     <NA>      QLD     576
5          5       1913 Hanoi Way          Nagasaki     463       35200
        phone         last_update
1             2006-02-15 09:45:30
2             2006-02-15 09:45:30
3 14033335568 2006-02-15 09:45:30
4  6172235589 2006-02-15 09:45:30
5 28303384290 2006-02-15 09:45:30
```

```r
# store table
dbGetQuery(con, "SELECT * FROM store LIMIT 5")
```

```
  store_id manager_staff_id address_id         last_update
1        1                1          1 2006-02-15 09:57:12
2        2                2          2 2006-02-15 09:57:12
```

```r
store_addresses <- dbGetQuery(con,
"SELECT *
FROM address
WHERE address_id IN (SELECT address_id FROM store);
")
```

```r
store_addresses
```

```
  address_id            address address2 district city_id postal_code phone
1          1  47 MySakila Drive     <NA>  Alberta     300
2          2 28 MySQL Boulevard     <NA>      QLD     576
          last_update
1 2006-02-15 09:45:30
2 2006-02-15 09:45:30
```

- **SELECT * FROM address**: Retrieves all columns (*) from the `address` table.
- **WHERE address_id IN (SELECT address_id FROM store)**: Filters the results to include only the rows in the `address` table where the `address_id` matches the `address_id` in the `store` table.

R-Rated Movie Stock Analysis by Store and Category

```
# Analyze R-rated movies
query2 <- "
WITH movie_stock AS (
  SELECT
    s.store_id AS store_id,                 -- Alias for store ID
    a.address AS store_address,             -- Alias for store address
    c.name AS category,                     -- Alias for movie category
    COUNT(i.inventory_id) AS total_dvds    -- Aggregating function to count DVDs

  FROM store s

  JOIN address a ON s.address_id = a.address_id

  -- Join 1: Link store to address

  JOIN inventory i ON s.store_id = i.store_id

  -- Join 2: Link store to inventory

  JOIN film f ON i.film_id = f.film_id

  -- Join 3: Link inventory to films

  JOIN film_category fc ON f.film_id = fc.film_id

  -- Join 4: Link films to film_category

  JOIN category c ON fc.category_id = c.category_id

  -- Join 5: Link film_category to category

  WHERE f.rating = 'R'
  -- WHERE clause to filter R-rated movies

  GROUP BY s.store_id, a.address, c.name

  -- Group by store, address, and category

)

SELECT * FROM movie_stock
ORDER BY store_address, category;
"
```

- **`WITH movie_stock AS (...)`**: A **Common Table Expression (CTE)** named `movie_stock` is created to temporarily store the intermediate results. This makes the query more organized and easier to understand.

Inside the CTE:

1. **`s.store_id AS store_id`**: Retrieves the unique ID of each store and labels it as `store_id`.
2. **`a.address AS store_address`**: Fetches the physical address of each store from the `address` table and labels it as `store_address`.
3. **`c.name AS category`**: Retrieves the name of the movie category (e.g., Action, Comedy) and labels it as `category`.
4. **`COUNT(i.inventory_id) AS total_dvds`**: Counts the number of DVDs (`inventory_id`) in each store for each category and labels the total as `total_dvds`.

The query combines data from multiple tables to include all necessary details:

- **`JOIN address a ON s.address_id = a.address_id`**: Links the `store` table to the `address` table to retrieve the store address.
- **`JOIN inventory i ON s.store_id = i.store_id`**: Links the `store` table to the `inventory` table to fetch the DVDs available in each store.
- **`JOIN film f ON i.film_id = f.film_id`**: Links the `inventory` table to the `film` table to get movie details.
- **`JOIN film_category fc ON f.film_id = fc.film_id`**: Links the `film` table to the `film_category` table to identify the category of each movie.
- **`JOIN category c ON fc.category_id = c.category_id`**: Links the `film_category` table to the `category` table to retrieve the name of the movie category.

- **Filter** (`WHERE f.rating = 'R'`): Only movies with an **R-rating** are included in the results.
- `GROUP BY s.store_id, a.address, c.name`: Groups the results by store ID, store address, and movie category to ensure that the count of DVDs is calculated separately for each unique combination.
- `SELECT * FROM movie_stock`: Retrieves all the columns (store ID, store address, category, and total DVDs) from the `movie_stock` CTE.
- `ORDER BY store_address, category`: Sorts the results alphabetically by store address and then by movie category for easier interpretation.

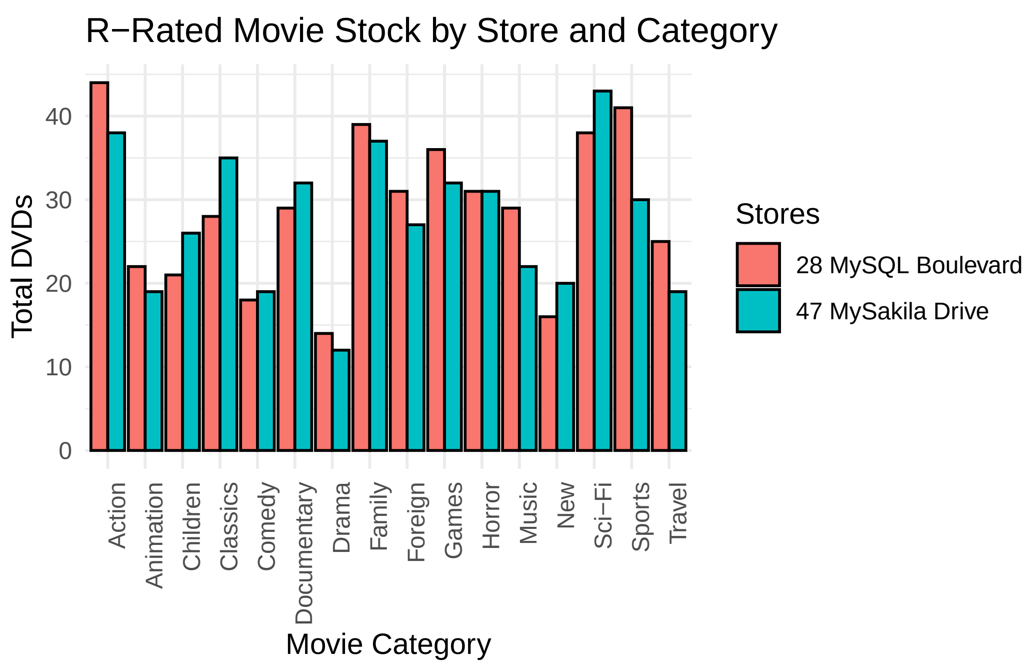
```r
movie_stock <- dbGetQuery(con, query2)
```

```r
head(movie_stock)
```

```
  store_id      store_address    category total_dvds
1        2 28 MySQL Boulevard      Action         44
2        2 28 MySQL Boulevard   Animation         22
3        2 28 MySQL Boulevard    Children         21
4        2 28 MySQL Boulevard    Classics         28
5        2 28 MySQL Boulevard      Comedy         18
6        2 28 MySQL Boulevard Documentary         29
```

```r
# Plot results
ggplot(movie_stock, aes(x = category, y = total_dvds, fill = store_address)) +
  geom_bar(stat = "identity", position = "dodge", color = "black") +
  labs(title = "R-Rated Movie Stock by Store and Category",
       x = "Movie Category",
       y = "Total DVDs",
       fill = "Stores") +
  theme_minimal() +
  theme(axis.text.x = element_text(angle = 90, hjust = 1))
```

```
Don't know how to automatically pick scale for object of type <integer64>.
Defaulting to continuous.
```



The bar chart shows the stock of R-rated movies in two stores: **28 MySQL Boulevard** and **47 MySakila Drive**. The movies are grouped by categories like Action, Comedy, Drama, and more. The height of each bar represents the total number of DVDs available in each category for the two stores. For example, Action and Sci-Fi have the highest stock across both stores, while categories like Children and Music , Drama have fewer DVDs. The chart makes it easy to compare the inventory of R-rated movies between the two stores and across different categories.

Execution Plan Analysis for R-Rated Movie Inventory Query

```
# Query execution plan
query3 <- "
EXPLAIN ANALYSE
WITH movie_stock AS (
  SELECT
    s.store_id AS store_id,                -- Alias for store ID
    a.address AS store_address,            -- Alias for store address
    c.name AS category,                    -- Alias for movie category
    COUNT(i.inventory_id) AS total_dvds    -- Aggregating function to count DVDs
  FROM store s
  JOIN address a ON s.address_id = a.address_id  -- Join 1: Link store to address
  JOIN inventory i ON s.store_id = i.store_id    -- Join 2: Link store to inventory
  JOIN film f ON i.film_id = f.film_id           -- Join 3: Link inventory to films
  JOIN film_category fc ON f.film_id = fc.film_id-- Join 4: Link films to film_category
  JOIN category c ON fc.category_id = c.category_id -- Join 5: Link film_category to categor
  WHERE f.rating = 'R'                            -- WHERE clause to filter R-rated movies
  GROUP BY s.store_id, a.address, c.name          -- Group by store, address, and category
)
SELECT * FROM movie_stock
ORDER BY store_address, category;
"
```

This SQL query analyzes the execution plan for the "**R-Rated Movie Stock Analysis** query" using `EXPLAIN ANALYSE`, which provides detailed information about the query's performance, including costs, timing, and the number of rows processed at each step.

- **`EXPLAIN ANALYSE`**: This prefix is used to execute the query and analyze its performance. It provides insights into the query's **execution plan**, including:

1. **Execution cost**: The estimated computational cost of each operation. It is expressed as a range: `cost=Start_Cost..End_Cost`. **Start Cost**: The cost to start the operation. **End Cost**: The cumulative cost after completing the operation.
2. **Actual time**: How long each step took to execute.
3. **Rows processed**: Number of rows involved in each step.

Most Expensive Operation by Cost: **Hash Join between `inventory` and `store`**

Key Evidence from the Query Plan:

1. **Cost**: The cost is **109.40 to 218.60**, making it the most expensive operation in the query. **Start Cost (109.40)**: The cost to initialize the join. **End Cost (218.60)**: The total cost after completing the join. This high cost indicates the operation is computationally intensive, consuming significant resources.
2. **Rows Processed**: **904 rows** are processed in this step. This includes rows passed from the `inventory` table (Step 11) and rows from the `store` table.
3. **Execution Time**: The step took **0.686 ms** to start and **1.521 ms** to complete. The time shows this is one of the more time-consuming steps in the query.

Why Is It Costly?

1. **Input Size**: The **`inventory` table** has **4,581 rows**, and the **`store` table** has only **2 rows**. Even though the `store` table is small, the operation involves processing every row in the `inventory` table to match it with the correct store.
2. **Hash Table Creation**: A **hash table** is created in memory for one table (likely `store`) to speed up matching. Each row from the other table (likely `inventory`) is scanned and checked against the hash table. While hash joins are efficient for small datasets, they become costly when the input size increases.
3. **Dependency on Previous Steps**: The inefficiency in **Step 11 (Sequential Scan on `inventory`)** means unnecessary rows are included in this join. For example: Step 11 scans all **4,581 rows** in the `inventory` table, but only rows relevant to R-rated movies should have been passed. This increases the workload for the hash join. The store table is small, but the inventory table has many rows. The database checks all inventory rows for matches, which takes more time because it didn't filter out unnecessary rows earlier.

```r
query_plan <- dbGetQuery(con, query3)
```

```r
View(query_plan)
```

```r
# Disconnect from the database
dbDisconnect(con)
```